Part 12

—

Topological Sorting

CSCI 3110 Code

Summer 2015

For topological sorting, we present two algorithms here, just as in class. Both algorithms are expected to produce the list of vertices of the given graph in topologically sorted order. The first algorithm uses DFS and exploits the fact that a topological ordering is nothing but a reverse postordering. The second one applies the approach of "peeling sources". Each algorithm has its own appeal. The DFS-based algorithm is easy to implement, given that we already have functions for DFS and postordering. However, it reverses the list of vertices in postorder, so it cannot produce the list of vertices in the topological ordering on the fly; the entire list must be produced in one go. It would be more in the spirit of lazy evaluation if every vertex in the topological ordering were produced as it is needed. The simple topological sorting algorithm based on peeling sources achieves this goal but has the "downside" of requiring the *ST* monad for its implementation, in order to keep track of the number of not-yet-peeled in-neighbours of each vertex.

1 DFS-Based Algorithm

Given that we already have a *dfs* function and a *postorder* function, topological sorting becomes trivial:

$$\begin{aligned}&\mathit{dfsTopSort} :: \mathit{AdjList}\ v\ \mathit{vl}\ \mathit{el} \to [V]\\&\mathit{dfsTopSort} = \mathit{reverse} \circ \mathit{postOrder} \circ \mathit{dfs}\end{aligned}$$

2 Peeling Sources

The simple topological sorting algorithm based on peeling sources is easily implemented using a recursive formulation. Since this algorithm needs to keep track of a mutable array containing the in-degrees of vertices, we implement it in the *ST* monad. Specifically, we run a computation *simpleTopSortM* on the graph *g*:

$$\begin{aligned}&\mathit{simpleTopSort} \quad :: \mathit{Show}\ v \Rightarrow \mathit{AdjList}\ v\ \mathit{vl}\ \mathit{el} \to [V]\\&\mathit{simpleTopSort}\ g = \mathit{runST}\ (\mathit{simpleTopSortM}\ g)\end{aligned}$$

simpleTopSortM g first collects the in-degrees of all vertices in *g* in an array *indegs* and uses it to identify all the sources of *g*. It then thaws *indegs* into a mutable array and applies *peel mIndegs* to the identified sources, which produces a vertex list starting with these sources, followed by vertices that become sources after removing the initial set of sources, and so on. This is exactly the strategy of our simple topological sorting algorithm from class:

```
simpleTopSortM   :: Show v ⇒ AdjList v vl el → ST s [V]
simpleTopSortM g = do let gs      = adjGraph g
                          indegs  = array (1, gNumVertices gs) $
                                      map (λv → (vIx v, vInDegree v)) (gVertices gs)
                          sources = filter (λv → indegs ! vIx v ≡ 0) (gVertices gs)
                      mIndegs ← thaw indegs
                      peel mIndegs sources
```

peel does the actual work of peeling the sources. For every source, it iterates over its out-neighbours, decreases their in-degrees by one, and collects those vertices that become sources as a result. It then returns the input list of sources followed by the result of recursively peeling the collected out-neighbours:

```
peel        :: STArray s Int Int → [V] → ST s [V]
peel indegs = go
  where go [ ] = return [ ]
        go vs  = do ws ← mapM peelVertex vs
                    (vs++) <$> go (concat ws)
```

peelVertex takes care of removing a single source and collecting its out-neighbours that become sources:

```
peelVertex v = catMaybes <$> mapM collectOutNeighbour (vOutNeighbours v)
```

collectOutNeighbour finally takes care of decreasing a given vertex's in-degree by one and returning *Maybe* this vertex if its in-degree is now 0:

```
collectOutNeighbour w = do let i = vIx w
                           d ← (subtract 1) <$> readArray indegs i
                           writeArray indegs i d
                           return (if d ≡ 0 then Just w else Nothing)
```